Towards a better understanding of testing if conditionals

Shimul Kumar Nath*, Robert Merkel†, Man Fai Lau* and Tanay Kanti Paul*
*Faculty of Information and Communication Technologies, Swinburne University of Technology, Hawthorn, Australia
Email: {snath,elau,tpaul}@swin.edu.au
†School of Information Technology, Monash University, Clayton, Australia
Email: robert.merkel@monash.edu

***Abstract*—Fault based testing is a technique in which test cases are chosen to reveal certain classes of faults. At present, testing professionals use their personal experience to select testing methods for fault classes considered the most likely to be present. However, there is little empirical evidence available in the open literature to support these intuitions. By examining the source code changes when faults were fixed in seven open source software artifacts, we have classified bug fix patterns into fault classes, and recorded the relative frequencies of the identified fault classes. This paper reports our findings related to "if-conditional" fixes. We have classified the "if-conditional" fixes into fourteen fault classes and calculated their frequencies. We found the most common fault class related to changes within a single "atom". The next most common fault was the omission of an "atom". We analysed these results in the context of Boolean specification testing.**

***Keywords*-fault types; bug fix patterns; fault frequency; bug analysis; fault-based testing;**

I. Introduction

Unit testing remains the most common method to verify the quality and reliability of the modules and subsystems which make up a complex software system. As such, the selection of the most effective unit testing techniques for the particular software under test is of much practical importance. There are many testing techniques that can be used to select test cases for unit testing, therefore, testers need to use their judgement to choose which technique or techniques they will use to select test cases.

One technique that software testers may apply is fault based testing. Fault based testing [1] is to choose test cases that can uncover specific fault types, if they exist, in the software under test. If the software passes the test cases, the testers can have confidence that those particular fault types do not exist in the software system. Of course, no testing method other than exhaustive testing can guarantee the absence of *all* faults; fault-based testing merely provides confidence that a specific class of faults is not present.

For fault based testing to be useful, testers must select a fault-based technique that reveals faults which are actually present in the software system. As the faults that actually present in the software are unknown before testing, testers must predict which fault types are likely to be present. The best way to inform such predictions is to collect empirical data about fault frequencies in real life software artifacts.

Recently Pan *et al.* [2] performed an empirical study of "bug fix patterns" - that is, patterns of source code changes seen when a bug is fixed - in seven open source Java artifacts. They analysed the differences between two incremental updated versions stored on the repositories of these individual projects. Pan *et al.* categorized the identified changes into various bug fix patterns. Their results show that there are two particularly common bug fix patterns: one is related to changes in method call parameters (e.g. `methodName(ownerName, amount)` changed to `methodName(accountId, amount)`) and the other is related to changes in the conditional expressions in programming constructs such as those in `if` conditionals (e.g. `"if (x > 4)"` changed to `"if (x > 4 && y < 3)"`). The method call parameter changes account for 14.9–25.5% of the defined bug fix patterns in the seven systems studied by them whereas the changes in the `if` conditional expressions are 5.6–18.6%. Given the high frequency of bugs related to `if` conditional expressions, and given that there are some existing fault based testing techniques for testing boolean specification (e.g. [3], [4]), it is worthwhile investigating how often a particular types of these Boolean expressions related faults are made by the software developers.

Pan *et al.* classified the `if` condition change (IF-CC) fix patterns into six categories. However this classification is somewhat ad-hoc, not orthogonal (i.e, some of the fix patterns matched more than one category) and does not align with any "pre-defined" fault classes (e.g. [5]). As a result, testers may not be able to use this information to choose appropriate fault specific testing techniques. In this study, we have devised a classification scheme of the `if` condition change patterns and calculated their relative frequencies, which, as well as being orthogonal, aligns as closely as possible to the fault classes of Boolean expression testing methods [5].

Our study was designed to provide testers additional information for choosing fault specific testing for `if` conditional faults. In this context, we aimed to answer the following research questions:

- **RQ1:** What percentage of IF-CC fix patterns can be classified automatically based on the fault classes of Boolean expressions?

- **RQ2:** What are the relative frequencies of the IF-CC fix patterns over several projects?

We believe that our results will enable software testers to make more informed decision when choosing testing methods. In particular, we wished to examine the effectiveness of existing Boolean expression testing techniques in revealing the major fault classes relating to the IF-CC fix patterns.

The paper is organised as follows. Section II describes some important definitions used in this paper. Section III describes the IF-CC fix patterns, LRF sub-patterns and non-fix patterns. Section IV presents the methodology that we applied to our research. Section V describes the tools implementation and verification. Section VI presents the extracted hunk data and frequencies of IF-CC fault classes. Section VII examines issues relating to cross project similarity as well as the effectiveness of specification based testing of `if` conditional. Section VIII discusses threats to validity. Section IX summarizes some related works. Section X reports our conclusions and suggestions for future works.

II. Terminology

Revision: a set of changes to the software source code grouped by the developer.

Hunk: a single contiguous or near-contiguous section of source code which has undergone a change from revision to revision. For our purposes, the division of the changes in a single revision into hunks is based on the rules of the Unix *diff* program.

Noise: a hunk which is not related to actually fixing a fault.

Non fix hunk: a noise that reflects changes to programming code that do not reflect any semantic changes.

Atom: a sub-condition in a single conditional expression. For instance, a conditional expression is `x>0||y<10`; it consists two sub-conditions- `x>0`, and `y<10`. Here each sub-condition is called as an atom.

III. Classification of IF-CC patterns

A. IF-CC fix patterns

Our intention is to examine whether existing Boolean expression testing methods are suitable for revealing common faults in real software systems, by assessing the fault frequencies of fixed bugs in real software projects. To collect this fault frequency information, we require a classification scheme which is automatically classifiable (to calculate the frequency), and aligned with the existing fault classes in the Boolean Expression fault (to select existing testing method). We have devised a scheme which is adapted from the fault classes for Boolean expressions as described by Lau and Yu [5]. We have proposed ten IF-CC fix patterns: Literal Reference Fault (LRF); Literal Omission Fault (LOF); Term Omission Fault (TOF); Literal Insertion Fault (LIF); Term Insertion Fault (TIF); Literal Negation Fault (LNF); Term Negation Fault (TNF); Expression Negation Fault (ENF); Operator Reference Fault (ORF); and Multiple Fault (MF). Table I shows the definitions and examples of these fault classes.

In Boolean specification testing, truth values are considered for a single atom of the conditional. However, a single atom in the conditional inside `if` statements may involve complex operations such as- arithmetic operations, method invocation, and comparison of objects. This makes it more challenging to generate appropriate test data than if they are just simple Boolean variables. Therefore, deeper knowledge about the fault of a single atom may help to generate suitable specification based test sets. Preliminary investigation revealed LRF faults are more common. We have therefore considered Literal Reference Faults (LRF) in more detail and classified them into five subclasses. The subclasses of LRF faults are Method Call Parameter Change (LRF-MCP), Method Call Change (LRF-MCC), Method Object Reference Change (LRF-MORC), Relational Operator Change (LRF-ROC), and Other Change (LRF:Other). Table II shows the definitions and examples of LRF sub-petterns.

B. IF-CC non-fix patterns

Preliminary manual examination of change patterns showed that, similar to Jung *et al.* [6] (see the result in VI-D), some common patterns of code changes to `if` conditionals which do not correspond to a code fix. We found six such patterns: Exchange Operands in Equals Method (NF-EOEM), Exchange Operands in an Infix Expression (NF-EOIE), Exchange Between Constants and Variables (NF-EBCV), Variable Name Changed (NF-VNC), Addition/Removal of Meaningless Parenthesis (NF-ARMP), and Refactoring Changes (NF-RC). These IF-CC non-fix patterns are described in Table III with an example. These non-fix patterns will be identified and removed from further consideration just before the final classification of the IF-CC fix patterns.Please refer to Section IV-C2 for details.

IV. Research Methodology

Software configuration management (SCM) data is a valuable resource for empirical software engineering research. For our study, we sought to identify and extract bug fix changes from the repository. Figure 1 shows the workflow for identifying, extracting and filtering the bug fix changes.

A. Identification, extraction and hunk pair generation of bug fix changes

1) Identification of bug fixes: There are two ways to identify the bug fix revisions from the SCM repositories as described by data mining researchers.

- *Keyword searching approach:* This approach for finding corrective maintenance from Version Control System (VCS) was first introduced by Mockus and Votta [7]. Log messages written by developers accompanying

Table I
IF-CC FIX PATTERNS AND THEIR CORRESPONDING EXAMPLES

<table>
<tr><th>IF-CC patterns</th><th>Example</th></tr>
<tr><td>Literal Reference Fault (LRF): A single atom change in an if condition</td><td>-if(data.getUsername() == null)
+if(getUsername() == null)</td></tr>
<tr><td>Literal Omission Fault (LOF):Addition of an atom with a leading logical AND (&&) in fix hunk</td><td>-if(source == caretStatus)
+if(source == caretStatus && evt.getClickCount() == 2)</td></tr>
<tr><td>Term Omission Fault (TOF): Addition of an atom with a leading logical OR (||) in fix hunk</td><td>-if(idealIndent == -1 ||
-(!canDecreaseIndent && idealIndent < currentIndent))
+if(idealIndent == -1 ||
+idealIndent == currentIndent || (!canDecreaseIndent &&
+idealIndent < currentIndent))</td></tr>
<tr><td>Literal Insertion Fault (LIF): Removal of an atom in fix hunk that was in bug hunk with a leading logical AND (&&)</td><td>-if(sorter != null && buffers.contains(buffer))
+if(sorter != null)</td></tr>
<tr><td>Term Insertion Fault (TIF): Removal of an atom in fix hunk that was in bug hunk with a leading logical OR (||)</td><td>-if(engine.isContextSensitive() ||
-"text".equals(buffer.getMode().getName()))
+if(engine.isContextSensitive())</td></tr>
<tr><td>Literal Negation Fault (LNF): Addition or removal of logical NOT (!) operator in fix atom which is being with a leading logical AND (&&)</td><td>-if(ref != null && !ref.isPrepared())
+if((ref != null) && ref.isPrepared())</td></tr>
<tr><td>Term Negation Fault (TNF): Addition or removal of logical NOT (!) operator in fix atom which is being with a leading logical OR (||)</td><td>-if(item == null || item.isStream())
+if(item == null || !item.isStream())</td></tr>
<tr><td>Expression Negation Fault (ENF): Addition or removal of logical NOT (!) operator in the whole fix expression</td><td>-if(!row.isEmpty())
+if(row.isEmpty())</td></tr>
<tr><td>Operator Reference Fault (ORF): Change of logical operator/s without changing any atom in fix expression</td><td>-if(palette != null || palette.length != 0)
+if(palette != null && palette.length != 0)</td></tr>
<tr><td>Multiple Fault: More than one fault scenario of nine fault classes are existing in the fix expression</td><td>-if(config.top != null && config.top.length() != 0)
+if(config == null)</td></tr>
</table>

Table II
LRF IF-CC SUB PATTERNS AND THEIR CORRESPONDING EXAMPLES

<table>
<tr><th>LRF sub-patterns</th><th>Example</th></tr>
<tr><td>Method Call Parameter Change (LRF-MCP): The bug atom and the fix atom are method call type and both contain the same method name but different parameter/s</td><td>-if(jEdit.getBooleanProperty("jdiff.horiz-scroll"))
+if(jEdit.getBooleanProperty(HORIZ_SCROLL))</td></tr>
<tr><td>Method Call Change (LRF-MCC): The bug atom and the fix atom are method call type but their method names are different</td><td>-if(!dummyViews.contains(window))
+if(!dummyViews.remove(window))</td></tr>
<tr><td>Method Object Reference Change (LRF-MORC): The bug atom and the fix atom are method call type but they are different by their object references</td><td>-if(data.getUsername() == null)
+if(getUsername() == null)</td></tr>
<tr><td>Relational Operator Change (LRF-ROC): The bug atom and fix atom use a different relational operator</td><td>-if(index > 0 && index < this.buffers.size())
+if(index >= 0 && index < this.buffers.size())</td></tr>
<tr><td>Other Change (LRF-Other): Any change to an atom not fitting into the above four categories</td><td>-if(drag)
+if(dragStart != null)</td></tr>
</table>

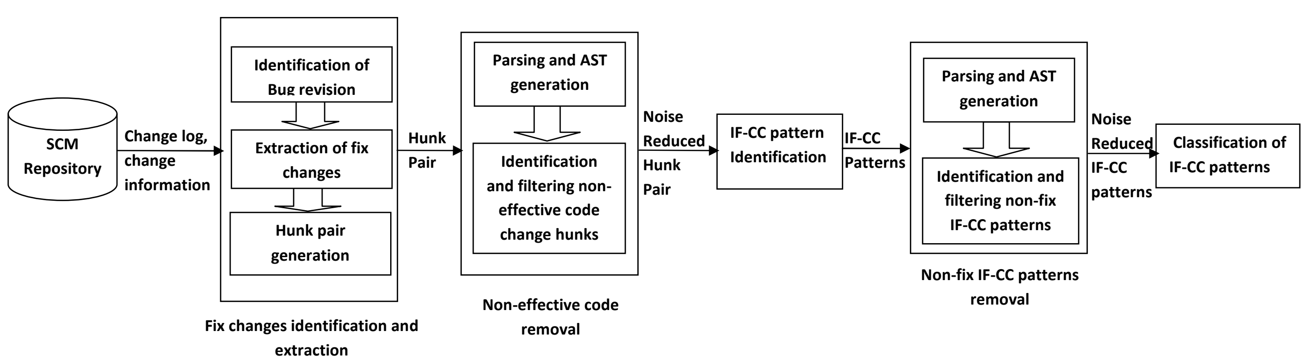

Figure 1. Work-flow of identification, extraction and filtering of bug fix changes

revisions to the source code, stored by the version control system, are scanned for keywords (such as bug, fix, fault, bug ID, and so on). The data thus collected is called VCS data. This approach has risks of including both false positives and false negatives. False negatives occur when developers do not include the keywords in

Table III
NON-FIX PATTERNS (NF) IN IF-CC AND THEIR CORRESPONDING EXAMPLES

Non-fix patterns	**Example**
Exchange Operands in Equals Method (NF-EOEM): The operands of an equals method in bug and fix atoms are exchanged	`-if(methodName.equals("finalize"))` `+if("finalize".equals(methodName))`
Exchange Operands in an Infix Expression (NF-EOIE): The operands of an infix expressions in such a way as to have no semantic impact	`-if((-1 != row) && (!table.isRowSelected(row)))` `+if(row != -1 && !table.isRowSelected(row))`
Exchange Between Constants and Variables (NF-EBCV): Exchange between constant and variable in fix atom, where the variable always contains the constant value	`-if((slist == null) || (slist.getChildCount() != 3))` `+if((slist == null) || (slist.getChildCount() != BODY_SIZE))`
Variable Name Changed (NF-VNC): Change the variable name in fix atom, where the values of the variables are always the same	`-if(cell.getHeight() > maxHeight)` `+if(cell.getHeight() > currentMaxHeight)`
Addition/Removal of Meaningless Parenthesis (NF-ARMP): Addition/Removal of parenthesis in fix atom or expression, which do not change the order of evaluation of the expression	`-if(before < 0 || Character.isWhitespace(line.charAt(before)))` `+if((before < 0) || Character.isWhitespace(line.charAt(before)))`
Refactoring Changes (NF-RC): Addition or removal of redundant constant comparisons, or type casts	`-if(mAutoFillEmptyCells == true)` `+if(mAutoFillEmptyCells)`

their log messages when fixing bugs. False positives can occur when, for instance, unrelated changes to source code are also included in a revision that contains a bug fix.

- *Linking approach:* This approach was proposed by Fisher *et al*. [8]. It establishes a link between bug tracking system (BTS) data and VCS data so as to get more accurate bug data from the available repository information. In a bug tracking system, all issues (not all which are bugs) are assigned a unique numeric identifier. In the linking approach, the VCS data is searched for issue ID to find the code changes corresponding to the issue. However, the links are imperfect; Ayari *et al*. [9] found that there were many missing links between the Bugzilla (BTS) and CVS (VCS) repositories for the Mozilla project. Only a fraction of BTS entries are actually bugs; Ayari *et al*. showed, by manual inspection of a subset of bug reports that only 58% issues in the BTS repository actually indicated corrective maintenance.

In this paper, we chose to use the keyword searching approach to identify the bug fix revisions due to the low identification rate of the linking approach.

2) Extraction of bug fix changes: Bug fix changes were extracted from the SVN repository using the `svn diff` command. Developers normally change different sections of a single or multiple files to fix a bug. The resulting "diffs" show the differences between the two revisions, showing different sections in a single file with a separator containing with the range information in Unidiff format [10]. For our purposes, each distinct section identified by diff was identified as a separate bug fix hunk that represents the bug fix change.

3) Hunk pair generation: Each bug fix hunk was then extracted out of the diff file. Hunk pairs are generated for each hunk with the help of the `patch`, applying the fix hunk to the previous revision of the file. These hunk pairs were described as the "bug hunk" and "fix hunk".

B. Non-effective hunk filtering

Removing noise - change hunks that are not actually code changes related to bug fixing - takes places at two stages in our workflow. The first class of noises filtered in our approach is "non-effective change hunks". These include changes that have no effect on the source code as perceived by the compiler - for instance, changes to comments. They are removed from further processing immediately after hunk pair generation, as shown in Figure 1.

To remove these hunks, we generate the parse trees including only the effective Java codes of the IF-CC hunk pairs (excluding, for instance, comments). If the "effective" trees for the code segments are same, we identify the hunk as non-effective and exclude it as a noise.

Elimination of effective non-fix hunks was done after identification of IF-CC patterns and is described in section IV-C2.

C. Analysis and pattern classification

Fluri *et al*. [11] proposed an algorithm to identify particular changes between two versions of a program. It compares every node of two ASTs (Abstract Syntax Tree) of the two revisions of a file to identify specific changes. As such two ASTs are to be generated for every hunk pair, and each node of the ASTs are to be compared to identify particular change in a fix hunk. In this study we adopt their approach to identify the IF-CC fix patterns.

1) Identification of IF-CC: The two generated ASTs are compared node by node. While any node difference is found and both nodes are `if` statements, our tool then considers whether the change is actually an `if` conditional change. Only the following cases were considered as a hunk contained an IF-CC pattern.

- While the *conditions* of the two `if` statements are not the same but their `then` statements are the same, this fix hunk is considered as IF-CC fix pattern.
- While the *conditions* of the two `if` statements are not the same and their `then` statements are not the same, this fix hunk can be considered or rejected as IF-CC fix pattern as follows -
  (i) If the `then` block was more than 5 statements long, and no more than two of these differed, it was considered as an IF-CC hunk pattern.
  (ii) If the `then` block was less than 5 statements and the differed statement/s are differed due to refactoring (such as a variable renaming, or changing a casting keyword), it was considered as IF-CC hunk pattern.
  (iii) All other cases were not considered IF-CC hunk patterns and these were ignored for further analysis.

2) Non-fix IF-CC reduction techniques: The second group of "noise hunks" are known as "Non-fix change hunk", which are code changes performed at the same time as a bug fix, which do affect the code's parse tree, but are not actually related to the bug fix. For instance, the addition of parentheses to an expression for readability. Non-fix IF-CC hunks were identified and eliminated, where this was possible. However, not all such non-fix hunks can be straightforwardly removed.

To perform this elimination, a case-by-case check was performed on each identified IF-CC change hunk to see whether the parse trees match patterns indicating a non-fix hunk. For instance, consider a case when a condition of an `if` is an `equals` method and the condition is changed in the fix revision by exchanging the operands of the `equals` method, leaving everything else unchanged. This is a non-fix change, as the change has no semantic impact on the program. For example, consider a case where `if(methodName.equals("finalize"))` is changed to `if("finalize".equals(methodName))`. Our tool's static analyzer checks whether both conditions are `equals` methods. If they are both `equals` methods, it compares the operand set of the `equals` method of the bug revision with the operands set of the `equals` method of the fix revision. In the example, the operand set for bug revision is {`methodName` , `"finalize"`} and the operand set for the fix revision is {`"finalize"`, `methodName` }. As the two operand sets are same, it is considered as NF-EOEM (Exchange Operands in Equals Method) and eliminated as noise. Among six identified non-fix IF-CC hunks (See in Section III-B), NF-EOEM, NF-EOIE, and NF-ARMP are eliminated as they are detected with static analysis. Other non-fix IF-CC patterns are detectable using some complex analysis techniques (such as- class analysis), which we did not implement in this work. The undetected non-fix IF-CC patterns affect the result by adding some false positive data. Those non-fix hunks which we did not eliminate were classified as LRF:other, thus inflating the frequency of this pattern by some amount. A manual check on a subset of data was conducted to determine the likely impact of this and the results are reported in Section VI-D.

3) IF-CC Pattern classification: The filtered IF-CC change hunks were then classified into the categories described in Table III. We used Fluri *et al.*'s [11] tree traversal algorithm as the basis of our classifier. The classifier compared each atom of the if condition in turn and used a simple decision tree approach to determine which pattern was applicable.

V. Tools Implementation and verification

A. Extractor

We have developed a module in Perl to extract the bug fix hunks automatically. It uses a two-step process:

- **Identify the fix revision:** The keyword searching approach described in Section IV-A1, is used to extract the bug fix revisions. Following the general approach of (for example) Ayari *et al.* [9], we used the following regular expression in our extractor to extract fix revisions:

  fix(e[ds])?|bugs?|patches?|defects?|faults?

- **Extract fix hunk:** Our tool extracted the change for a specific bug fix revision using the *svn diff* command and the identified revision number. From this change information we have separated different regions of changes as different hunks.

After extracting the fix hunks, hunk pairs are generated using the Unix *patch* command using hunks and their corresponding revision file. To verify the output of the extractor module, we manually checked all the hunk pairs of *Checkstyle*, one of the seven artifacts studied in this study.

B. Analyser

We have developed an analyser module in Java that makes use of the Eclipse JDT Tooling [12] to identify the IF-CC fix patterns. The module takes a hunk pair as input and generates two ASTs. These two ASTs are compared node-to-node to see whether differed nodes in two ASTs are both `if` condition or not. While the differed nodes are `if` condition and the *condition* of two if statements are not same but *then* statements are same, this fix hunk is extracted as IF-CC fix pattern. Other differed nodes with `if` conditions changes were flagged and checked manually to determine whether they were IF-CC hunks. To verify the correctness

of the Analyser, we manually checked all the identified IF-CC patterns of *Checkstyle* and *JabRef* by the analyser.

C. Classifier

We have developed a classifier module in Java, again utilizing the Eclipse JDT Tooling, for classifying the IF-CC fix patterns identified by the Analyser according to the described fault classes in Section III. To verify the correctness of the Classifier, we have manually verified all the classified IF-CC patterns of all seven artifacts.

VI. Results

A. Software Artifacts

We have selected seven open source systems implemented in Java for our study. All seven artifacts have a long term maintenance history. We have considered change histories ranging from 4 years to 10 years, depending on the available revisions in `SVN` repositories. We selected open source software projects because we have easy access of their repositories. We have extracted the bugs from the incremental revisions of the software, not just the bugs found in customer releases. As unit testing methods are often applied to incremental versions as well as major ones, we believe it is appropriate to consider bugs in incremental revisions as well as major releases. Moreover, it also provides a much larger sample to work with.

B. Artifact properties and extracted bug fix data

Table IV shows the considered maintenance histories, the number of revisions, extracted bug fix revisions, extracted fix hunks, effective fix hunks and extracted IF-CC hunks for the software artifacts have found in their corresponding `SVN` repositories. The result suggests that neither the rates of revisions over time nor the rates of extracted bug fix hunks are consistent in different software artifacts. This should be kept in mind when considering cross-project comparisons.

We have analysed the effective fix hunks through our analyser, and identified the IF-CC fix hunks. Table IV shows that the IF-CC fix hunk frequencies varied from 4.3%, in *DrJava* to 10.7%, in *PMD*. Despite the variation in relative frequency in each artifact, we had sufficient examples of IF-CC fix patterns to collect meaningful statistics about their properties for all seven artifacts.

C. Frequencies of different IF-CC fix patterns

Table V shows the numbers and frequencies of different IF-CC fix patterns along with the detected non-fix patterns across the seven software artifacts. It shows that the percentage of different fault classes in seven software projects are different. However, it is clear that the rank of frequencies of different fault classes are quite consistent over the seven artifacts.

LRF is the most common fault class in all the projects, although its frequency varies from 35.0% in *FreeCol* to 84.4% in *PMD*. The mean frequency for LRF is more than 50% of total IF-CC faults. In other words, over half of all `if` condition faults are related to a single condition inside the `if` expression. The second and third largest frequency groups are LOF and TOF respectively in all the artifacts, except in *PMD* where their rankings were reverse. Thus almost one-third of the `if` condition faults are due to omission of subconditions inside the `if` expression. The other fault classes (LIF, TIF, LNF, TNF, ENF, ORF) had frequencies of between 0 to 4% in the different software projects, which indicates that these fault classes are much less frequent than the major three class of faults. Besides those "single fault" classes, a considerable proportion of all faults are *multiple faults*, representing between 4.9% to 19.2% of all faults across all the software artifacts. Multiple faults are the most next common fault type with the `if` expression after LRF, LOF and TOF.

From Table V, the LRF subcategories vary substantially from project to project. Among them, either the LRF:Other or the LRF:MCC are the major LRF sub-classes in the seven projects. The LRF:MCP and LRF:MORC represent 3.7% to 9.4% and 1.5% to 18.1% respectively of total faults related to IF-CC patterns. LRF:ROC faults were significant in some projects but not others, with their frequency varying between 0.4% to 4.8% of all IF-CC faults.

Some of the non-fix patterns that described in Section III-B are detected by our tool and have been summarized in the Table V. The results suggest that non-fix pattern detection rates are not the same across different projects. Perhaps, neither all the non-fix patterns are present nor the frequencies are the same in all the software artifacts. As a result, some of the non-detected non-fix IF-CC patterns in our result are considered and are classified as LRF:Other.

Table VI
FREQUENCIES OF NON-FIX IF-CC PATTERNS IN OUR SAMPLE SET

Project Name	Total Hunk	Non-fix Hunk	Our tool Identified
Checkstyle	92	13 (14.1%)	7
JabRef	106	5 (4.7%)	0
iText	108	5 (4.6%)	0
DrJava	100	9 (9.0%)	3
jEdit	110	3 (2.7%)	2
Total	**516**	**35 (6.8%)**	**12**

D. Manual study to trace non-fix hunks

The noises with the non-code changes described in IV-B are removed from our data. In a previous general study of fault pattern data, Jung *et al*. [6] identified 11 common non-fix patterns, which represented almost 8.3% of the total hunks in their manual inspection. However, none of the 11 non-fix patterns related to if-condition changes. While this gave us some indication that most IF-CC fix hunks did indeed represent effective code changes, we still considered

Table IV
ANALYSED PROJECTS AND THEIR CHANGE DESCRIPTIONS

Project Name	Considered Period	No. of Revisions	No. of fix Revisions	Total hunks	Effective hunks	IF-CC hunks	% of IF-CC
Checkstyle	20/06/2001-15/08/2010	2538	609	3404	1685	92	5.5%
JabRef	14/10/2003-21/08/2010	3316	413	4131	2032	106	5.2%
iText	28/11/2000-15/08/2010	4576	506	2813	1063	108	10.2%
DrJava	19/06/2001-30/08/2010	5419	1208	38014	11781	511	4.3%
jEdit	01/07/2006-17/08/2010	18382	3767	34651	19904	1027	5.2%
PMD	21/06/2002-22/09/2010	7141	988	10186	4444	475	10.7%
FreeCol	14/01/2002-30/09/2010	7484	1431	13571	3226	215	6.7%

Table V
CLASSIFIED IF-CC PATTERNS IN DIFFERENT FAULT CATEGORIES AND THEIR FREQUENCIES

Fault category	Checkstyle		JabRef		iText		DrJava		jEdit		PMD		FreeCol	
LRF:MCC	6	7.1%	4	3.8%	3	2.8%	164	32.8%	159	15.7%	274	58.7%	7	3.3%
LRF:MCP	5	5.9%	7	6.7%	4	3.7%	29	5.8%	67	6.6%	44	9.4%	14	6.5%
LRF:MORC	3	3.5%	19	18.1%	4	3.7%	37	7.4%	34	3.4%	7	1.5%	5	2.3%
LRF:ROC	4	4.7%	5	4.8%	4	3.7%	6	1.2%	24	2.4%	2	0.4%	6	2.8%
LRF:Other	26	30.6%	23	21.9%	50	46.3%	137	27.4%	295	29.1%	67	14.3%	43	20.1%
Category total	**44**	**51.8%**	**58**	**55.2%**	**65**	**60.2%**	**373**	**74.6%**	**579**	**57.1%**	**394**	**84.4%**	**75**	**35.0%**
LOF	18	21.2%	19	18.1%	16	14.8%	51	10.2%	164	16.2%	18	3.9%	54	25.2%
TOF	10	11.8%	13	12.4%	6	5.6%	25	5.0%	108	10.7%	28	6.0%	30	14.0%
LIF	0	0.0%	0	0.0%	2	1.9%	10	2.0%	41	4.0%	1	0.2%	6	2.8%
TIF	1	1.2%	3	2.9%	3	2.8%	3	0.6%	22	2.2%	1	0.2%	3	1.4%
LNF	0	0.0%	0	0.0%	0	0.0%	3	0.6%	0	0.0%	1	0.2%	0	0.0%
TNF	0	0.0%	0	0.0%	1	0.9%	0	0.0%	0	0.0%	0	0.0%	1	0.5%
ENF	0	0.0%	1	1.0%	1	0.9%	5	1.0%	8	0.8%	0	0.0%	4	1.9%
ORF	0	0.0%	1	1.0%	2	1.9%	0	0.0%	3	0.3%	1	0.2%	0	0.0%
Multiple	12	14.1%	10	9.5%	12	11.1%	30	6.0%	89	8.8%	23	4.9%	41	19.2%
Analysed Fault	92		106		108		511		1027		475		215	
Total Fault	85		105		108		500		1014		467		214	
Non-fix	7		1		0		11		13		8		1	

Table VII
NON-FIX IF-CC PATTERNS FOUND IN OUR SAMPLE SET

Non-fix IF-CC patterns	Checkstyle	JabRef	iText	DrJava	jEdit	**Total**
NF-EOEM	5	0	0	0	1	**6**
NF-EOIE	0	0	0	0	1	**1**
NF-EBCV	3	0	0	0	0	**3**
NF-VNC	3	3	4	6	1	**17**
NF-ARMP	2	2	0	3	0	**5**
NF-RC	0	5	1	0	0	**3**
Total	**13**	**5**	**5**	**9**	**3**	**35**

it prudent to conduct a manual check a representative set of IF-CC fix hunks, to identify and classify non-fix IF-CC patterns (if they exist). All extracted IF-CC patterns of *Checkstyle*, *JabRef*, *iText* and randomly selected sets from other two artifacts (*DrJava* and *jEdit*) were inspected manually for this study. We have analysed a total 516 IF-CC fix patterns selected from five software systems (*Checkstyle*, *JabRef*, *iText*, *DrJava* and *jEdit*) and found 35 non-fix IF-CC patterns, which is almost 6.8% of the analysed IF-CC hunks (see in Table VI). We have classified them into six categories which is described in Section III-B.

Table VII shows that the frequencies of six non-fix IF-CC patterns are not similar in different software artifacts. Only one non-fix pattern, Variable Name Changed (NF-VNC), among six was found in all software artifacts. The NF-VNC was the most common non-fix pattern seen, representing almost half of the total non-fix patterns observed. The frequencies of IF-CC non-fix patterns in five software artifacts varies from 2.7% to 14.1%, which indicates their inconsistent appearance in different software artifacts (see Table VI). Perhaps, this would happen due to different software applications having different code complexity or, different programmers choosing different coding styles. For instance, some programmers use redundant parentheses to increase the code readability whereas some programmers do not.

VII. DISCUSSION

A. Cross project frequency similarity

We found that most of the IF-CC fix patterns can be classified based on the fault classes of specification based testing of Boolean expressions, which is the answer of our first research question (RQ1). To answer the second research question (RQ2), we have calculated the Spearman's rank correlation coefficient [13] of ten major IF-CC fix patterns for all the seven projects (see Table VIII) and found most of the correlation values are higher than 0.8 (except two values). The statistical significance of these 21 correlation values are calculated using the Holm-Bonferroni

Table VIII
SPEARMAN RANK CORRELATION VALUES OF IF-CC FOR SEVEN ARTIFACTS

Checkstyle	JabRef	iText	DrJava	jEdit	PMD	FreeCol	
1.00	0.94	0.94	0.85	0.89	0.86	0.86	Checkstyle
	1.00	0.92	0.78	0.88	0.81	0.80	JabRef
		1.00	0.81	0.93	0.85	0.87	iText
		0.81	1.00	0.93	0.79	0.96	DrJava
				1.00	0.84	0.95	jEdit
					1.00	0.73	PMD
						1.00	FreeCol

Table IX
SPEARMAN RANK CORRELATION VALUES OF LRF SUB PATTERNS FOR SEVEN ARTIFACTS

Checkstyle	JabRef	iText	DrJava	jEdit	PMD	FreeCol	
1.00	0.10	0.22	0.60	0.90	0.80	0.90	Checkstyle
	1.00	0.89	0.00	0.30	-0.10	0.30	JabRef
		1.00	-0.22	0.22	-0.22	0.45	iText
			1.00	0.80	0.90	0.30	DrJava
				1.00	0.90	0.80	jEdit
					1.00	0.60	PMD
						1.00	FreeCol

method [14], with an overall $\alpha = 0.05$. We have found all 21 correlations are statistically significant. We can conclude that the ranking of IF-CC fix patterns is truly similar across projects.

We also calculated the Spearman correlation coefficients for the sub-patterns of LRF across the seven software artifacts (see Table IX). The statistical significance of these correlations are calculated using Holm-Bonferroni method [14], and found that the correlations were not significant with an overall $\alpha = 0.05$. These result suggests that the appearance of LRF subpatterns are not consistent over projects.

B. Implications for Testing

As discussed in Section VI-C, the three most common fault categories are

1) LRF ranging from 35.0% to 84.4%
2) LOF ranging from 3.9% to 25.2%
3) TOF ranging from 5.6% to 14.0%

In the seven artifacts, the fraction of all if-condition faults falling in to these categories ranges from 74.2% (FreeCol) to 94.3% (PMD), which can be calculated from Table V. Hence, any testing methodologies which are good at detecting LRF, LOF and TOF (for example, the MUMCUT testing techniques [4]) would be very useful in revealing almost $\frac{3}{4}$ of the IF-CC bug fixes. Hence, if software testers are to perform testing given limited time and resources, our results suggest that selecting those testing techniques that can reveal LRF, LOF and TOF would be a reasonable choice. The high correlation of fault category frequency across the seven artifacts (as discussed in Section VII-A), therefore, suggests that this recommendation is generally applicable of testing `if` conditionals across different types of Java software systems.

Furthermore, as suggested by the fault class hierarchy [5], test cases that can detect those at the lower part of the hierarchy (e.g. LRF, LOF and TOF) will be able to detect those corresponding faults on the upper part of the hierarchy (e.g. TNF, LNF, ORF and ENF). Hence, these test cases have a good chance to detect faults categorized as LNF, TNF, ORF and ENF.

VIII. THREATS TO VALIDITY

A. Internal Validity

The *Extractor* identified bug fix revisions using the keyword searching approach (described in section IV-A1). Developer omission of these keywords in log messages, or the inclusion of unrelated changes in a revision tagged as a bug fix, will result in false negative and false positives respectively. False negatives are a less significant concern, as there is no obvious reason to suspect that untagged bugs would have markedly different fix patterns than tagged ones. However, false positives could well affect the relative frequencies of fault categories.

The *Analyser* can detect and eliminate only a fraction of all non-fix IF-CC patterns. However, the non-fix IF-CC hunks which have been considered in our analysis, are mostly classified in LRF:Other or broadly in LRF. As a result, the actual frequency may fluctuate with our calculated frequency for LRF. In general, given the pattern of our results and the results of our manual analysis, we do not believe that the non-eliminated non-fix IF-CC patterns have substantially affected our conclusions.

B. External validity

We have restricted our data set by selecting software systems implemented in Java which may not be representative for other software projects that are developed in other programming languages. Since different programming

languages have different constructs, the frequencies of different IF-CC fix patterns frequency may vary for different programming languages.

We have consciously selected well maintained software artifacts, with wide usage, which may not reflect the bug fix patterns in less well maintained and widely used open source software systems.

Our artifacts are all open source software. Proprietary artifacts may, or may not, show difference in frequencies of IF-CC fix patterns with our counted frequencies. This suggests to look at proprietary software for future work.

IX. Related works

Historical information and relative frequencies of fault types can help practitioners to select suitable testing methods. From the testing researchers point of view, such historic information and relative frequencies may allow the general recommendation of particular unit testing techniques, or serve as inspiration to devise newer and more effective testing methods. A number of researchers ([15], [16]) have devised techniques using historical information to identify the most fault prone files or modules of a software project. These types of techniques are helpful to reduce testing effort by predicting mostly fault prone files, however, they do not provide much information about how to test them. An attempt has been taken by Hayes in [17], where a number of fault classification studies are used to analyse the merits of various testing techniques in object oriented software. But no relative frequency has been considered and the classification was largely based on the author's personal testing experience.

There are several attempts to devise effective fault classification scheme for a number of different purposes (such as improving debugging). One well-known fault classification effort was by Knuth [18] who classified the errors found or reported in ten years of the development of his well-known typesetting software, TEX into nine categories. The errors were classified manually by Knuth based on his own logs and recollections. TEX is a somewhat unusual software system that has been developed by Knuth himself, largely alone, to satisfy a specification he himself devised. This classification is neither convenient to replicate without manual developer involvement, nor provides sufficient information to guide white-box testing.

Static checking tools, such as FindBugs [19], automatically classify some bug patterns. These patterns indicate bugs which occur due to mistakes with code idioms, or misuse of language features. Those bug fix patterns can be detectable by using static checking tools. As testing researchers, we are primarily interested to look for bug fix patterns which demand testing rather than static checking to detect them - static checkers should be used to find and remove such bugs as can be identified *before* testing!

DeMillo and Mathur [20] devised an alternative classification scheme based on semantic changes of bug fixing for the purpose of strengthen debugging and choosing suitable testing but their study was limited to a single software artifact. In the recent time, Pan *et al*. [2] devised an automatic syntactic classification scheme for the better understanding about bugs and their frequency characteristics, and justified the scheme over seven open source Java software projects (not the same as ours). Nath *et al*. [21] replicated this classification scheme on a single additional open source Java software artifact and suggested some potential improvements to the classification. They also described the necessity of exploring more specific information for specific bug fix patterns to determine *how* best to test for revealing the most common patterns.

In this study, we have devised an orthogonal fault classification scheme and calculated their frequencies over seven open source Java projects. We have considered implications of this result in the context of testing `if` conditionals.

X. Conclusion and Future Works

The contributions of our paper are 14 automatically extractable `if` conditional bug fix patterns and their relative frequencies over seven open source Java software artifacts. We calculated the statistical significance by computing rank correlation and found that the frequencies of 10 major IF-CC fix patterns are highly correlated over seven software projects. The classification is orthogonal and shares properties of "pre-defined" fault classes of existing specification based testing. Considering the difficulties of generating specification based test sets for `if` conditionals, we have subcategorised the LRF into five sub-patterns. However, we did not find significant cross-project correlation among the LRF sub-patterns. We also have identified a set of non-fix patterns in IF-CC fix patterns. We have accounted the fault frequency information for better understanding of testing `if` conditionals.

There are some obvious opportunities for extension of this work. Most straightforwardly, similar methodologies could be applied to other conditional constructs such as `for`, `while`, and `do-while` loops. It is unknown, but certainly a relevant question, to see whether the fault category frequency is similar in loop conditions and if-conditions. There are specific patterns which might reasonably be suspected to be more common in loop conditions, such as changes to operators or a single atom to fix fencepost errors, so empirical data to support this long-held supposition would be of considerable practical interest.

The relatively common Multiple Fault (MF) pattern detection is also interesting for further study. Some research has examined (e.g. [22], [23], [24]) specification-based testing for double faults in Boolean specifications. However, the present study does not provide a clear indication of the proportion of multiple faults - which may involve more than

two faults - can be detected. Further research is required to determine whether there are such appropriate testing techniques for the most common multiple fault types.

Finally, many other procedural and object-oriented programming languages offer similar conditional constructs, but they may differ in their detailed syntax and implementation. It would be interesting to see whether there is any difference in fault category frequencies in projects implemented in different programming languages.

If fault based testing is to be effective, we believe it must be based on a solid empirical understanding of the faults that are actually present in real-world software systems. This paper represents a step in that direction.

References

[1] L. J. Morell, "A theory of fault-based testing," *IEEE Transactions on Software Engineering*, vol. 16, pp. 844–857, August 1990.

[2] K. Pan, S. Kim, and E. J. Whitehead Jr., "Toward an understanding of bug fix patterns," *Empirical Software Engineering*, vol. 14, no. 3, pp. 286–315, August 2008.

[3] J. J. Chilenski and S. P. Miller, "Applicability of modified condition/decision coverage to software testing," *Software Engineering Journal*, vol. 9, no. 5, pp. 193 –200, September 1994.

[4] Y. T. Yu, M. F. Lau, and T. Y. Chen, "Automatic generation of test cases from Boolean specifications using the MUMCUT strategy," *Journal of Systems and Software*, vol. 79, no. 6, pp. 820–840, June 2006.

[5] M. F. Lau and Y. T. Yu, "An extended fault class hierarchy for specification-based testing," *ACM Transactions on Software Engineering and Methodology*, vol. 14, no. 3, pp. 247–276, 2005.

[6] Y. Jung, H. Oh, and K. Yi, "Identifying static analysis techniques for finding non-fix hunks in fix revisions," in *DSMM '09: Proceeding of the ACM first international workshop on Data-intensive software management and mining*. New York, NY, USA: ACM, 2009, pp. 13–18.

[7] A. Mockus and L. G. Votta, "Identifying reasons for software changes using historic databases," in *ICSM '00: Proceedings of the International Conference on Software Maintenance (ICSM'00)*. Washington, DC, USA: IEEE Computer Society, 2000, p. 120.

[8] M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," in *ICSM '03: Proceedings of the International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 2003, p. 23.

[9] K. Ayari, P. Meshkinfam, G. Antoniol, and M. Di Penta, "Threats on building models from CVS and bugzilla repositories: the mozilla case study," in *CASCON '07: Proceedings of the 2007 conference of the center for advanced studies on Collaborative research*. New York, NY, USA: ACM, 2007, pp. 215–228.

[10] The Open Group, "Diff description, the open group base specifications issue 7," http://pubs.opengroup.org/onlinepubs/9699919799/utilities/diff.html, access date: 2011-03-17.

[11] B. Fluri, M. Wuersch, M. PInzger, and H. Gall, "Change distilling:tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 33, pp. 725–743, 2007.

[12] "Eclipse Java Development Tools (JDT) (project page)," http://http://www.eclipse.org/jdt/, access date: 2011-03-21.

[13] "Spearman's rank correlation coefficient," http://en.wikipedia.org/wiki/Spearman's_rank_correlation_coefficient, access date: 2011-02-04.

[14] S. Holm, "A simple sequentially rejective multiple test procedure," *Scandinavian Journal of Statistics*, vol. 6, pp. 65–70, 1979.

[15] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller, "Predicting faults from cached history," in *Proceedings of the 29th international conference on Software Engineering*, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 489–498. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2007.66

[16] T. J. Ostrand and E. J. Weyuker, "Software fault prediction tool," in *Proceedings of the 19th international symposium on Software testing and analysis*, ser. ISSTA '10. New York, NY, USA: ACM, 2010, pp. 275–278. [Online]. Available: http://doi.acm.org/10.1145/1831708.1831743

[17] J. H. Hayes, "Testing of object-oriented programming systems (OOPS):a fault-based approach," in *Object-Oriented Methodologies and Systems, volume LNCS 858*. Springer-Verlag, 1994, pp. 205–220.

[18] D. E. Knuth, "The errors of TeX," *Software-Practice and Experience*, vol. 19, no. 7, pp. 607–685, July 1989.

[19] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *ACM SIGPLAN Notices*, vol. 39, no. 12, pp. 92–106, December 2004.

[20] R. A. Demillo and A. P. Mathur, "A grammar based fault classification scheme and its application to the classification of the errors of TeX," Software Engineering Research Center; and Department of Computer Sciences; Purdue University, Tech. Rep., 1995.

[21] S. K. Nath, R. Merkel, and M. F. Lau, "An analysis of fault classification scheme for Java software," Center for Software Analysis and Testing; and Faculty of ICT; Swinburne University of technology, Tech. Rep., May 2010. [Online]. Available: http://www.swinburne.edu.au/ict/research/sat/technicalReports/TC2010-002.pdf

[22] M. F. Lau, Y. Liu, and Y. T. Yu, "On detection conditions of double faults related to terms in Boolean expressions," in *Proceedings of COMPSAC 2006: the 30th Annual International Computer Software and Application Conference*, Sep. 2006, pp. 403–410.

[23] ——, "On the detection conditions of double faults related to literals in Boolean expressions," in *Proceedings of 12th International Conference on Reliable Software Technologies - Ada-Europe 2007*, ser. LNCS, no. 4498, Ada-Europe. ,, June 2007, pp. 55–68.

[24] ——, "Detecting double faults on term and literal in Boolean expressions," in *Proceedings of 7th International Conference on Quality Software (QSIC 2007)*, Oct. 2007, pp. 117–126.